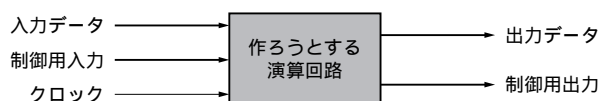
第2章

演算回路設計のセンスをつかもう

――演算をハードウェア化する際にどんなことを考えるか

森岡澄夫

仕様で求められる演算処理の内容を理解できるようになったら，今度は実際に回路に落とし込む作業に入ります．本稿では，回路化にあたって使われる一般的な考え方や設計のコツを説明します．ほとんどの実用回路は，それらのいずれか，もしくは複数を組み合わせることで作られています．（著者）

入力データ → 作ろうとする演算回路 → 出力データ
制御用入力 → 　 → 制御用出力
クロック →

1．どのくらいの性能にしたいのか？
- ターゲット・デバイスは？
- クロック周波数は？
- データのビット幅は？
- データ到来の間隔は？
- 処理のレイテンシ（遅延）は？
- 回路規模は？
- 消費電力は？

2．接続先はどのような回路なのか？
- CPUバスとつなぐのか？
- マスタ？　スレーブ？
- ほかのIPコアと直接つなぐのか？
- インターフェースの端子やプロトコルは？
- 起動と停止は誰が制御するのか？

3．設計の環境は？
- 何人で何カ月で作るのか？
- どの程度の品質を保証するのか？
- どんなツールを使って作るのか？

回路本体を作ってからつじつまを合わせようとしても駄目！
作る前に決める

図1　演算回路の本体を作り出す前に大ざっぱでも決めるべきこと
演算回路本体をさっさと作って安心したくなるものだが，その気持ちをグッと抑えて，まずは目標性能や接続先との交信方法について決めることが大事．もちろん，システム開発初期でこれらを確定的には決められなかったり，開発中に変更が入ったりすることも多いが，何も決めないよりは「後から回路を全面作り直し」のリスクをはるかに低減できる．

① まずはシステム全体のことを考える

意外に思われるかもしれませんが，演算回路だからといって数値計算本体から設計に入っていくわけではありません．どちらかといえばそれは後回しです．

● 回路内部よりも先に目標性能やI/F仕様を考える

回り道のようでも，システム全体の中で回路がどう使われるかを考え，回路の目標性能や入出力インターフェースをはっきりさせるところから入ります[(15),(16)]（**図1**，**図2**）．これは，ソフトウェアを複数人で分担して作成するような場合に，例えば関数の中身をいきなり作り始めたりせず，

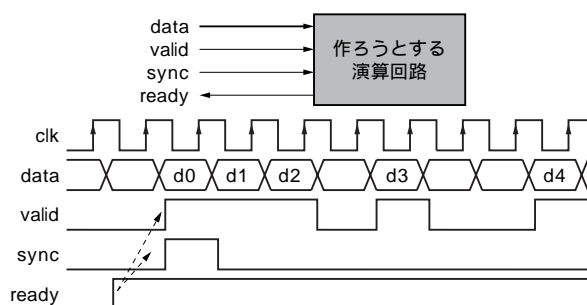



図2　データ転送プロトコル（タイミング・チャート）を決めた例
プロトコル設計というと難しそうだが，要はデータをやりとりする端子とタイミング・チャートを決めるということ．必ずしもこの図通りとは限らない．

KeyWord　64点高速フーリエ変換，FFT，テイラー展開，オンザフライ計算，離散コサイン変換，パイプライン，インターリーブ，抽象度

入出力仕様をまず決めるのと全く同じことです．

すなわち，

- どのくらいの性能の回路にしたいのか．加えて，使えるクロックの周波数や本数はどの程度で，どれくらいの回路規模なら許されるのか，どのようなデバイスを使うのか（FPGAかASICか．ASICなら何nmの製造プロセスか）
- 回路に対するデータの入出力（タイミング・チャートや，入出力のビット幅など）をどのようにするか
- どのくらいの設計期間なのか

などについて，システム設計者や隣接モジュールの担当者とよく打ち合わせます．これらの事項が最初から明確に分からない場合や，設計途中で変更が入るような場合もありますが，たとえ大ざっぱでも決めることが重要です．このことが演算回路本体をどのような方向性で作るかに，非常に大きく影響するからです．

また，何も考えずに演算回路本体だけ先に作ると，周辺回路とうまくつながらない，性能を全く発揮できないといった深刻な問題が後で発生しやすくなります．例えば，1クロックに1個，必ず入力データが来るという想定で必要性能が得られるよう演算アルゴリズムなどを選定したのに，実システムでデータ・レートが大幅に低いことが後で発覚したりすると，アルゴリズム変更が必要になって設計が完全にやり直しになってしまうかもしれません．

● 設計コンテストの着眼点

ところで，「Design Wave設計コンテスト2007」の課題は64点高速フーリエ変換（FFT）回路ですが[(3)]，この場合も単に汎用FFT回路の最適化ばかりに目を向けては面白くありません．FFTがどのような用途で使われており，その用途ではどれくらいの演算精度や回路性能が必要なのかを調べてみましょう．何か面白い特定用途向けに，ユニークな計算アルゴリズムや回路構成を考えてみると，際だった特色や高い実用性を示すことができるに違いありません．必ずしも型通りにバタフライ演算を行わなくても，本稿で説明するような色々なアイディアを適用する余地が出てくるからです．

② 数式を“そのまま”実装しなくてもよい

目標性能やインターフェースのタイミングが見えてきたら，仕様の計算内容（数式）をどのようなアルゴリズムを使って作るかを考えます．このときの基本的な発想の仕方は，誤解を恐れずに言えば，

- 数式をそのまま正直に計算しなくてもよい
- 数学的に複雑な計算を組むのは，なるべく避ける（バグのもと）
- 精度保証が必要だったり例外処理の多い計算方法は，なるべく避ける（テストが大変）

など，「設計の労力や回路コストをいかに少なく済ませるか」ということです．

例えば，FFT回路の数式，

$$X(k)=\sum_{n=0}^{N-1}x(n)\cdot e^{-j(2\pi/N)nk} \cdots\cdots(1)$$

をもし素直に計算するならば，まず$(2\pi/N)nk$を計算し，次に$e^{-j(2\pi/N)nk}$を計算し，それを繰り返すという手順になりそうです．また，2次元離散コサイン変換（DCT）の計算式，

$$F(u,v)=\left(\frac{2}{N}\right)^{\frac{1}{2}}\left(\frac{2}{M}\right)^{\frac{1}{2}}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\Lambda(i)\Lambda(j)\cdot\cos\frac{\pi u(2i+1)}{2N}\cdot\cos\frac{\pi v(2j+1)}{2M}\cdot f(i,j) \cdots(2)$$

ただし，$\Lambda(\xi)$は$\xi=0$のとき$1/\sqrt{2}$，それ以外のとき1

であれば，cos関数を例えばテイラー展開などを使って計算していくのかもしれませんし，開平計算も要りそうです．

しかし通常は，以下に述べるような方法のもとで，もっと簡単かつ確実に回路が作れないか，もっと回路性能が得られないかという点について検討を行います[注1]．

● オンザフライ計算と事前計算を使い分ける

回路実装に限らずソフトウェア実装でも頻繁に使われる方法ですが，難しい演算についてはオンザフライ計算（回路動作時にリアルタイムで計算すること）を行わず，設計時に事前計算した結果をテーブルで持っておくという手法があります（**図3**）．例えば離散コサイン変換のcos関数の値など，テーブルで持つことにするだけで設計はグッと楽になります．別途テーブル値を作成・検証する手間はかかりますが，込み入った回路を作るよりは楽です．

テーブルは，HDLの`case`文で書いて，論理合成で組み

注1：一般的な傾向として，算術演算関数を数学の教科書にある定義・定理通りに計算すると，回路が著しく複雑になる，性能が低下する，精度保証が難しくなる，といった問題が起こりがち．数学の教科書ではそういった工学上のコスト概念をあまり扱わないのが，一つの理由．

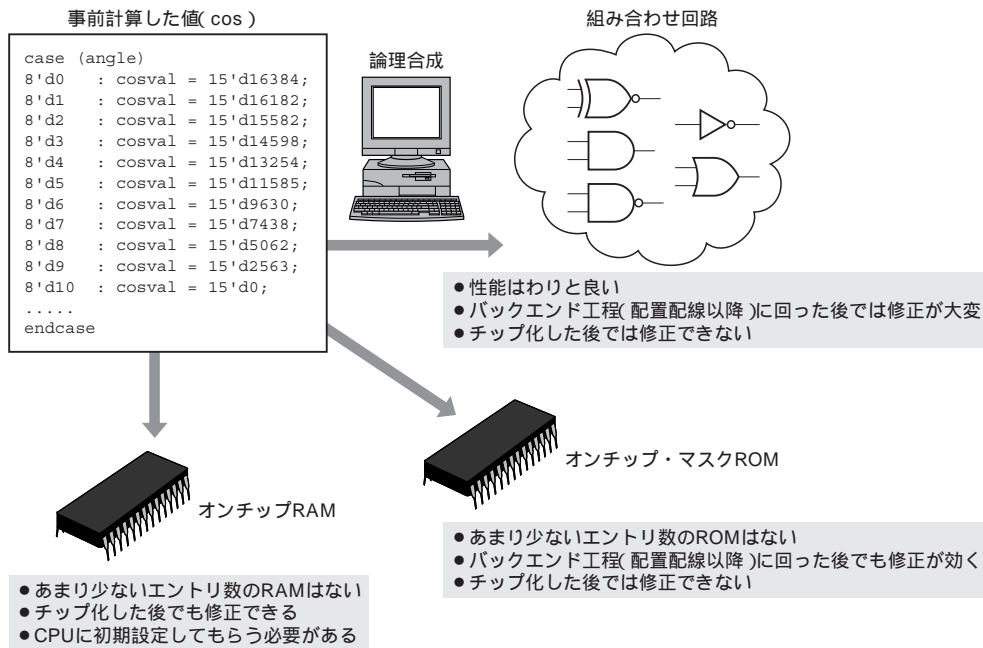



図3 事前計算結果の実装形態

事前計算したテーブルを使うことで回路構成の簡単化や回路作成の手間の削減が可能．テーブルの実装の仕方はいくつかあり，一長一短．

合わせ回路化したり，RAMやレジスタ(フリップフロップ)を用意してCPUに値をセットしてもらう，などの実装方法があります．テーブル値は0～数クロックで取り出せます．ただし，テーブル利用には次に示す注意点もあります．

- 一つのテーブルに持てる要素数はたかだか2^{12}～2^{16}個程度．気軽に利用できる目安は2^8個くらいまで
- オンザフライ計算した方が，回路性能が良い場合も少なくない(論理演算の塊のような場合など)
- オンザフライ計算とテーブル利用のどちらが性能上有利かは，ソフトウェアの場合と必ずしも同じではない．特に，ソフトウェアと異なり，事前計算したからといって速くなるとは限らない(ハードウェアでは複数の計算を並列実行できるため)

また実設計では，オンザフライ計算とテーブルを組み合わせたり，複数のテーブルを組み合わせたりすることも珍しくありません．参考文献(1)，(11)～(14)の高速除算回路などが面白いので，参照してください．

● 数体系・数表現を変換して演算を簡単にする

数体系を変換することで，同じ演算でもかなり簡単かつ効果的に行えるようになる場合があります(**図4**)．これも回路のみならず普遍的に使われている考え方です．また数体系とまで言わなくても，同じ数値をどのように2進表現するかも回路性能に大きな影響を与えます．

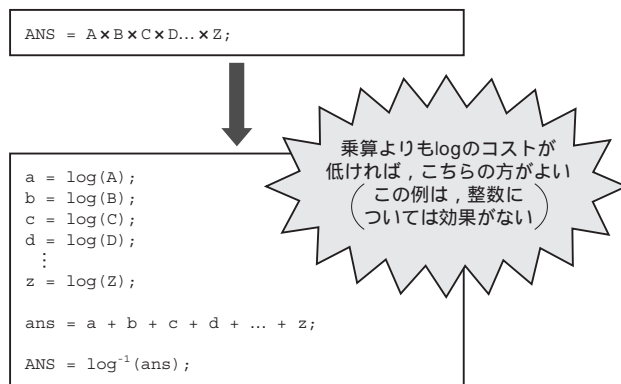



図4 数体系や数表現を変換して処理を簡単にする

図の例は，データが整数ならばあまり効果がないものの，有限体など整数以外では有効な場合がある．logが変換関数である．図と同じ考えのもと，なんらかの変換を行ってから本来やりたかった演算を行うことはよくある．

そのような例は無数に知られていますが，例えば，

- FFTなどによって時間領域から周波数領域に移すことで，周波数制御，データ圧縮，データ特徴抽出などを容易にする(FFTは，それ自体が目的というより，これら多くの用途に使われる1部品)
- 画像の座標系変換やハフ変換などにより，回転などの画像操作や特徴検出を容易にする
- 整数において，基数を変換したり冗長2進表現などを用いたりして演算コストを低減する．RSA演算におけるモンゴメリ変換[(7),(8),(9)]も同様の目的
- 有限体やベクトル・データなどにおいて，基底変換を

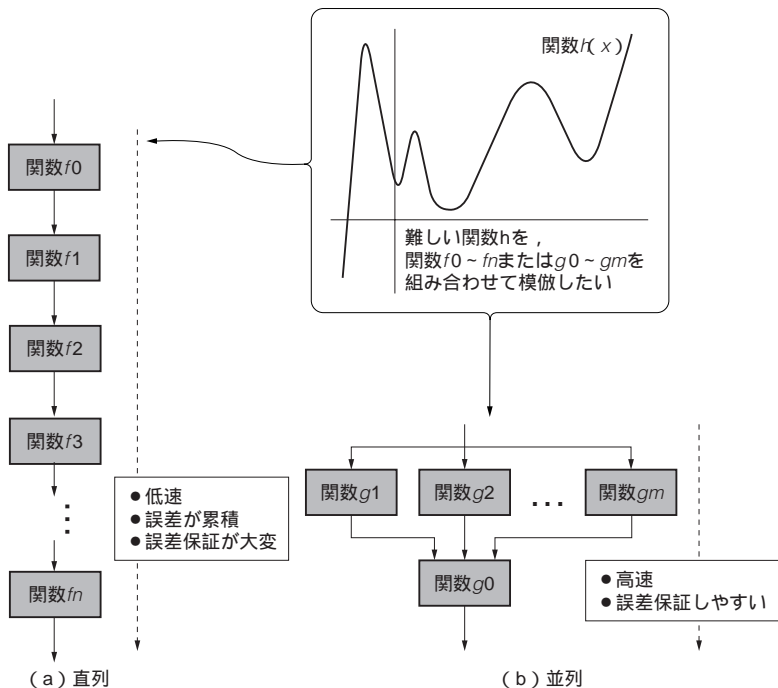



図5　近似計算するときは用いる関数を並列に並べる
一般に，関数を直列にすればするほど，速度(レイテンシ)が遅くなるとともに精度保証も難しくなる．

```
固定小数点数  newval, oldval;

/* 値が収束するまでループ */
while (abs(newval - oldval) > threshold) {
      oldval = newval;
      newval = 丸め(演算関数(oldval));
}

return (newval);
```

【問題点】
データによってループ回数が異なるので
●最悪処理時間を保証しにくい
●パイプライン化など並列化を図りにくい
●誤差を保証しにくい
（ループにより丸め誤差が累積・拡大）

図6　収束アルゴリズム(漸近法)による計算はなるべく避ける
ニュートン・ラプソン法やテイラー展開など，データが収束するまで繰り返し計算を行うようなタイプの処理は，図に示した問題点を解決できない限りは使用を避けたほうが無難．

行って演算コストを低減する[5]

などが挙げられます．効果の程度を一般的に言うのは難しいですが，演算速度や回路規模がけたオーダで劇的に改善するようなことはしばしばあります．

なお，変換・逆変換にはオーバヘッドが付きものです．そのため，変換によって回路コストが増加する場合もあるので[注2]，それでも変換のメリットがあるかを見極める必要があります．

● 近似計算する

高度な関数をその通り計算するのでなく，多項式関数などを組み合わせて近似計算する方法もしばしば使われます[11]〜[14]．近似に用いる関数群を，なるべく直列ではなく並列に並べるようにするのが工夫のしどころです(**図5**)．回路を高速化したり精度を保証するという点で好都合だからです．

この方法では，近似の方法にもよりますが，精度保証が難しい問題になることがあります．特にテイラー展開やニュートン・ラプソン法など，繰り返し計算で値を収束させていくアルゴリズム(漸近法)も一種の近似計算ですが，回路化には次のような問題点があります．解決できない限りは避ける方が無難です(**図6**)．

- 繰り返しによって，途中結果の保存に必要なビット数が極端に増加していくことがある(例えば乗算を繰り返した場合など)
- 回路コストを抑えるため，あるいは循環小数を打ち切るために途中結果を丸めると，今度は最終結果の精度保証が難しくなる．繰り返し処理によって丸め誤差が拡大・累積するが，その様子を解析するのは一般には簡単でない
- 入力データ値に依存して処理時間(繰り返し回数)が変わるので，インターフェースのタイミング設計や，データ処理の最低性能(値が収束するまでの最大繰り返し回数)の保証が難しい

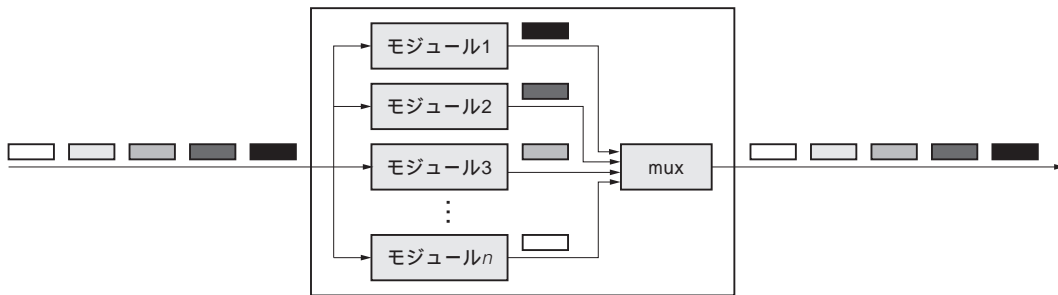
● 計算量の少ないアルゴリズムを使う

元の数式通りに計算するのではなく，より計算量(オーダ)の少ないアルゴリズムを使う方法があります．これも演算処理ごとに多数の研究・調査がなされています．例えば，第1章で述べたFFTについては参考文献(3)を，リー

注2：例えば，FFTをかけることによってデータは複素数になるし，変換前後でのデータのビット幅などにも慎重な気配りが必要になる．変換・逆変換時に丸めなどを行うなら，情報がロスしないかにも注意が必要．

(a) パイプライン処理の導入(時間並列)

(b) インターリーブ処理の導入(空間並列)

図7
データを連続的に投入できるようにするのが回路高速化の常とう手段
計算量の少ないアルゴリズムを使ったとしても，一つデータを入れたら処理が終わるまで完全に「だんまり」になってしまう回路では，それほどの処理性能は出ない．データを連続投入できるようにする方法は大きく分けて二つある．一つは(a)に示すパイプライン化で，直列につながった処理において，データを次の処理に回したらすぐ次のデータを受け付けるようにしたもの．もう一つは(b)のインターリーブ化で，処理モジュールを並列に並べて，空いているモジュールにデータを投入するようにしたもの．

ドソロモン符号については参考文献(6)を，RSA暗号については参考文献(7)～(9)を手がかりにして調べてみてください．

ただし，計算量が少ないと回路が高速になる(あるいは回路が小規模になる)，とは単純には言えないので注意が必要です．計算量は，ソフトウェアのように「演算を1ステップずつ逐次(順番に)実行したときの時間注3」を表す指標なので，並列処理が可能で演算コストも異なるハードウェアには当てはめられないのです(6)．詳細については誌面の都合上，解説を割愛します．

● ソフトウェアの高速化手段を使えないことがある

計算量の少ないソフトウェア用アルゴリズムでは，次のような高速化手段が頻繁に使われています．

- メモリ上に複雑なデータ構造(木構造，各種リスト，テーブルなど)を構築する
- それを動的に頻繁に更新する
- 関数の再帰呼び出し

これらの手法は普通のRTLとして記述できません．また，トリッキーな方法で無理に回路化しても，それほどの高速化につながるわけではありません．これも詳しい説明は割愛しますが，参考文献(2)などを参照してください．

● 高速化したければデータを連続処理できるようにする

もし，データ処理の高速化が必要なのであれば，計算量を減らすことよりも，データを連続投入できるような処理構成を考えることから検討するのが先決です．いろいろな粒度注4でパイプライン化を図る，インターリーブなどによる処理並列化を図る，などが最も基本的な手法です．**図7**にそれらの概要を示します．

具体的な回路がもう少し見える例として，1クロックに1データという速度を達成した1次元DCT回路の例を**図8**に示します(DCTではこれが標準的な回路構成(10))．発想はインターリーブと似ています．まず，シリアルにやってくるデータをn並列処理することで，1回路ユニット当たりのデータ速度を$1/n$に落とし，その後にパラレル-シリアル変換でデータ・レートを元の速度(1クロックに1データ)に引き上げています．

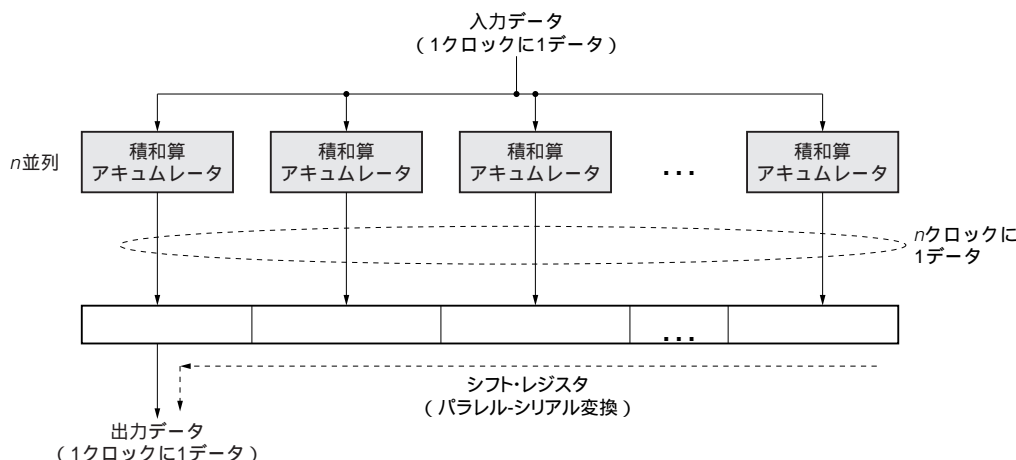
● 演算順序やデータ・アクセス順を工夫する

やや細かいテクニックですが，数式の計算回路では次の

注3：厳密には時間だけではなく，領域(メモリ)の話もあるが，ここでは割愛する．いずれにしても，計算量というのは基本的にチューリング・マシン上の概念であるから，回路のように自由なアーキテクチャ上で処理を実行できる場合には，当然，当てはまらなくなる．

注4：回路処理や回路構造などの「細かさ」のこと．直感的な例を示すと，個々の論理演算よりは整数演算の方が処理の粒度が大きい．論理ゲートの組み合わせよりは，ブロック図の方が構造の粒度が大きい．

図8
データの連続投入が可能な1次元離散コサイン変換回路の例

図は1次元DCT回路の標準的な構成法である．詳細は省略するが，ポイントは二つ．一つは計算処理を並列化しやすいアルゴリズムを使うこと．もう一つはパラレル-シリアル変換によってデータ幅を広くする代わりにデータ速度を落としたり，シリアル-パラレル変換によってデータ幅を狭くする代わりにデータ速度を上げたり，という速度調整を行うこと．例えると，川の水の流れの調節と同じようなことである．

ような方法が基本的な最適化法としてよく使われます[2]（**図9**）．

- 式を展開した後に因数分解するなど，式を変形して重い演算を減らす
- 共通項を抜き出して繰り返し計算を減らす
- 式の深さが減るよう式変形して高速化する
- 複数の式の間で演算器を使い回す（やり過ぎは禁物）

ゲート・レベルの最適化は粒度がより細かいのですが，考え方は多くの点で同じです．

$A×B+A×C$ ➡ $A×(B+C)$

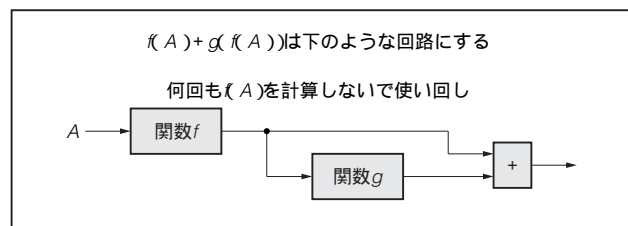
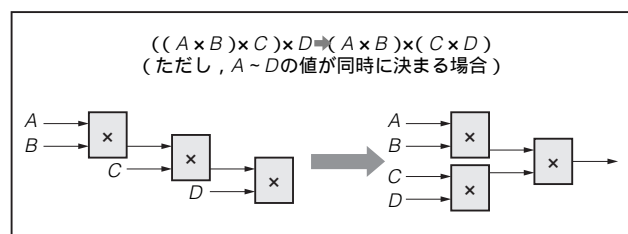
（**a**）式を変形して重い演算（乗算など）を減らす

（**b**）同じ計算の繰り返しをなくす

（**c**）式変形して深さを減らす

図9　数式を回路化する上での小技

図に示したのは，いずれも非常によく知られた常とう手段である．組み合わせ回路だけでなく順序回路でも適用できるが，そのときは高速化と回路規模のトレードオフの問題がより鮮明に出てくる．

また，回路へのデータの入力順（データのアクセス順）の工夫も，回路規模の削減や速度の向上に効果があります．ごく単純な例としては次のようなものが挙げられます．

- 入力データをためずに到着しだい処理を進められるよう，回路のインターフェース・プロトコルを設計する．例えば，受け取ったデータのCRC値を確認するときなど[17]，送信時にCRCを計算したときと同じ順番でデータを流してもらうようにする．そうしないと，多量のバッファ・メモリが必要になる（多くのECCでも同じ）
- 画像処理において，画面走査順などを工夫して画素参照回数を減らせる場合がある

● 適切な抽象度を選択して設計する

一般論としては「設計はなるべく高い抽象度で行う方がよい（例えばゲート・レベルよりはRTL，RTLよりはビヘイビア・レベル）」と言われますが，それはあくまで十分な性能を達成できる場合の話です．ハードウェア設計，特に演算回路設計においては，高い抽象度では必要な性能が出ない場合が頻繁にあります．

しかも，高い抽象度であればどんな処理でも簡単に作れるのかというと，そうでもありません．高い抽象度のプリミティブ（関数や記述法）しか使えない状況では，低い抽象度なら簡単に済む処理の記述が，非常に回りくどくなる場合があるからです．

例えば画像加工では，「あるしきい値より暗いところではフィルタ値Aを，明るいところではフィルタ値Bを適用」というアダプティブな処理などを，自分専用にきめ細かく工夫して使うことがよくあります．ところが，モデリング・ツールには「全画面に同一フィルタをかける」ような一

有限体GF(2^4)の乗算器のVerilog HDL記述．
ゲート・レベルだが規則性もあり容易に作れる．
出来上がる回路も高性能．

```
assign a0b0 = a[0] & b[0];
assign a0b1 = a[0] & b[1];
…（AND記述13行省略）…
assign a3b3 = a[3] & b[3];

assign c0 = a0b0;
assign c1 = a0b1 & a1b0;
assign c2 = a0b2 & a1b1 & a2b0;
assign c3 = a0b3 & a1b2 & a2b1 & a3b0;
assign c4 =        a1b3 & a2b2 & a3b1;
assign c5 =               a2b3 & a3b2;
assign c6 =                      a3b3;

assign o[0] = c0 ^ c4;
assign o[1] = c1 ^ c4 ^ c5;
assign o[2] = c2      ^ c5 ^ c6;
assign o[3] = c3           ^ c6;
```

（a）論理演算を使って組んだ回路

論理演算を一切使わずに作ろうとすると，それほど簡単には書けない．
剰余算なども必要になって回路性能も大幅悪化．

```
case (a) /* べき表現に */
4'd0:  ap = 4'd0;
4'd1:  ap = 4'd0;
4'd2:  ap = 4'd1;
4'd3:  ap = 4'd4;
  ⋮
4'd15: ap = 4'd12;
default: ap = 4'd0;
end case
```

```
case (b) /* べき表現に */
4'd0:  bp = 4'd0;
4'd1:  bp = 4'd0;
4'd2:  bp = 4'd1;
4'd3:  bp = 4'd4;
  ⋮
4'd15: bp = 4'd12;
default: bp = 4'd0;
end case
```

```
assign ad = (ap + bp) % 15;
```

```
case (ad) /* ベクトル表現に */
4'd0:  o1 = 4'd1;
4'd1:  o1 = 4'd2;
4'd2:  o1 = 4'd4;
4'd3:  o1 = 4'd8;
  ⋮
4'd14: o1 = 4'd9;
default: o1 = 4'd0;
end case

assign o = (a == 0 || b==0) ? 0 : o1;
```

（b）整数演算だけで実現

図10　「高い抽象度でコードを書いた方が簡単」とは限らない

図は有限体乗算の例．一般には「ゲート・レベルの記述は難しく，RTL，ビヘイビア・レベルと抽象度が上がるほど記述は簡単である」と信じられている．しかし，現実は必ずしもそう単純ではない．高い抽象度では，整数演算や処理のコントロール（forやifなど）などは確かにすっきり書ける．ところが，少しでも記法の範囲にない処理を書こうとすると，高い抽象度である点が突如として厄介な足かせになってしまう．ポイントは，高い抽象度で回路を書くことではなく，適切な抽象度を選んで書くことである．

般ニーズの高いプリミティブ関数しか用意されないのが普通で，それを使って上記処理を組むのはとても面倒なことになります注5．少し抽象度を下げてC言語などで組めば，きめ細かい処理を書けるので造作もありません．

さらに回路に近い例として，**図10**にエラー訂正符号などでよく使われる，有限体の乗算回路の例を示します(5)．**図10(a)**は論理演算を使って普通に組んだ回路です．それほど複雑なものではありませんし，規則性もあるので作成はいたって簡単です．しかし，これをソフトウェアのように整数演算だけで書こうとすると，**図10(b)**のように「有限体から整数へ数値体系を変換し，整数上で演算を行い，その結果を有限体へ逆変換する」という面倒なことになります（実際にC言語による実装ではよく見かける記述）．回路性能は**図10(a)**とは比較にならないほど悪化します．

注5：例えば，全画面XにフィルタAをかけた結果aとフィルタBをかけた結果bを両方作り，しきい値によるXの2値化結果Yを作り，$aY+bY$を計算する，など．回りくどいことこのうえないし，うまく書けないこともしばしば．

本稿では，演算処理回路の設計において，おおよそ普遍的に使われている考え方を紹介しました．もちろん現実には，設計対象や会社の事情などによって，さまざまな工夫や考え方があるはずです．それは実務を通してつかんでいくのがよいでしょう．筆者の経験としては，演算回路設計は書籍や本稿(!?)を読んだり，人の話を聞いて分かったつもりになっていても駄目で，実際に自分の手でやってみることが肝心です．

一見つまらなそうに見えることが，案外難しかったり重要であったりするからです．

参考・引用*文献

(1) 鈴木昌治；ディジタル数値演算回路の実用設計，CQ出版社，2006年11月．（演算回路の実用的設計手法）
(2) 森岡澄夫；HDLによる高性能ディジタル回路設計，CQ出版社，2002年11月．（回路設計のベースとなるいろいろな考え方）
(3) 和田知久；64点高速フーリエ変換回路設計仕様書，Design Wave Magazine，2006年11月号，pp.143-155，CQ出版社．（FFTの入門記事，Design Wave設計コンテスト2007の案内）
(4) 森岡澄夫，高野光司，大庭信之；IPコアの設計時に立ちはだか

る壁，Design Wave Magazine，2002年8月号，pp.120-126，CQ出版社．（設計時の抽象度の使い分け）
（5）森岡澄夫；エラー訂正や暗号処理で使われる演算回路を極める，Design Wave Magazine，2003年7月号，pp.57-67，CQ出版社．（有限体の回路設計）
（6）片山泰尚，森岡澄夫；ハードウェアで高速処理を実現したリードソロモン復号アルゴリズム，Interface，2000年7月号，pp.164-174，CQ出版社．（リードソロモン符号の超高速ハードウェア・アルゴリズム）
（7）佐藤証，森岡澄夫；暗号処理のソフトv.s.ハード，Design Wave Magazine，2003年9月号，pp.72-79，CQ出版社．（RSA暗号の処理回路）
（8）高野光司，大庭信之，森岡澄夫；C++を用いた公開かぎ暗号回路の細分化・階層化設計事例，Design Wave Magazine，2002年10月号，pp.148-157，CQ出版社．（C++からRSA暗号回路のRTLを導き出す）
（9）高野光司，大庭信之，森岡澄夫；暗号回路の処理速度と設計効率を向上させるには，Design Wave Magazine，2002年12月号，pp.151-159，CQ出版社．（RSA暗号回路の設計トレードオフなど）
（10）Latha Pillai；Video Compression Using DCT，Application Note：Virtex-II Series（XAPP610），Mar. 3，2005．http://direct.xilinx.com/bvdocs/appnotes/xapp610.pdf（DCTの易しい説明と典型的回路例）
（11）畔津明仁；割り算回路，Design Wave Magazine，1999年12月号，pp.115-126，CQ出版社．（除算器を例に，どのように演算回路の構成を決定していくか詳説）
（12）畔津明仁；高精度の逆数（1/x）を計算する，Design Wave Magazine，2000年5月号，pp.116-121，CQ出版社．（精度についてどう扱っていくか詳説）
（13）鈴木昌治；ASIC開発における数値演算回路設計の定石，Design Wave Magazine，2003年7月号，pp.24-38，CQ出版社．（演算器設計全般，近似計算による演算回路設計）
（14）鈴木昌治；漸近法を用いた除算回路，Design Wave Magazine，2005年12月号，pp132-138，CQ出版社．（演算精度をどう評価していくか）
（15）S.Mori；演算回路に不可欠なインターフェースの設計，Design Wave Magazine，2005年3月号，pp.56-69，CQ出版社．（IPコアのインターフェース設計の入門）
（16）森岡澄夫；システム全体を見渡しながら回路設計を行う，Design Wave Magazine，2006年5月号，pp.49-63，CQ出版社．（システムへIPコアを組み込む）
（17）森岡澄夫；わかる！ CRC計算回路の作り方とアレンジ，Interface，2004年12月号，pp.183-193，CQ出版社．（CRC計算回路の原理と設計）

もりおか・すみお
日本電気（株）システムデバイス研究所

COLUMN 実用演算回路をテープアウトするまでの道のり2
～基本設計書のレビューからチップ完成まで～

（第1章のコラムからのつづき．）要求仕様をきちんと理解し，回路の全体像がまとまったら，次のような工程に進みます．

手順6）基本設計書をレビューする

どのような処理を実現する回路を作るか，どのようなインターフェースにするか，どのようなブロック構造にするか，などをドキュメント化して，依頼者をはじめ関係者とレビューします．ここでの徹底的な確認が非常に重要です．回路規模や必要なコンポーネント（オンチップRAMやカスタム・マクロなど）の予測も伝えます．

手順7）設計環境や検証環境を構築する

EDAツールのセットアップ作業のみならず，どのような戦略で設計や検証を進めるかをよく相談して決定します．設計や検証の専用ツールを作る場合もしばしばあります．

手順8）RTLを書いて，まず設計者が検証する

詳細は省略します．一通りRTLが出そろうまでの期間は，案外短いこと（例えば1週間～1カ月など）が多いです．

手順9）コーディングの結果をレビューする

設計チームのメンバで，ソース・コードを実際に見て問題をチェックします．

手順10-1）論理合成などを通しバックエンドに回す準備を行う

ここからが実に長い道のりになります．論理合成で動作周波数が出ない，回路規模が大き過ぎる，RTL対ゲートの等価性検証がうまくできないなど，慎重に設計したつもりでいても問題はたくさん起きるものです．なお，論理合成などはいつも設計者自らが行うとは限らず，別部門や別会社が行うこともあります．

手順10-2）検証プランをレビューする

これまでの作業に並行して，検証者の方では検証プランを立てて検証仕様書を書き，自動検証環境（リグレッション可能）なども作っています．これをレビューします．

手順10-3）検証者に引き渡して念入りにテスト

シミュレータだけでなくFPGAなども導入し，とにかく念入りに検証を行います．チップ化する回路の検証で流すパターン数は，数百万～数億といったオーダです（それでも完全ではない）．バグ発見時の修正・再検証も含めて，完了までに数カ月かかることはざらにあります．

手順10-4）システムに統合して全体検証

IPコアをシステム（チップ）全体に組み込んでの検証も進めます．シミュレーションでは手に負えず，巨大なFPGAボードを使うことがよくあります．

手順11）バックエンド部隊に引き渡す

配置配線などの作業チームに設計データを引き渡します．ただし，上記の検証作業が完全に終っていなくても，「仮データ」として何度か事前に流すこともあります．ATPG（チップの製造テスト用のデータ生成ツール）でテスト・カバレッジが足りない，配置配線後に動作周波数が悪化した，など相変わらず問題は出てきます．

手順12）チップが完成する（本当に怖いのはここから）

チップが出来上がって製品に搭載されて世の中に出ていくのは，とてもうれしい体験です．反面，もしバグが発見されたらと思うと冷や汗が出てきますが，筆者は幸運にもその経験はありません．